Parallel Discrete-Event Simulation on Distributed-Memory Multicomputers

Seng Chuan TAY and Yong Meng TEO
Department of Information Systems
and Computer Science
National University of Singapore
Lower Kent Ridge Road
Singapore 0511
email: taysengc,teoym@iscs.nus.sg

Abstract

As computers become more powerful and their use expands, the need to simulate larger and more complex systems in reasonable computing times becomes more important. Parallel simulation can significantly speedup the process. Nevertheless, simulating complex systems on high-speed computers with multiprocessor capabilities is not a trivial task. This paper describes the parallel simulation of multi-station queueing networks on distributed-memory multicomputers. A deadlock-free conservative scheme for organizing parallel simulation is proposed, and implemented on a network of workstations running the PVM software. Preliminary results show that close-to-linear performance speedup can be achieved when the simulated system can be partitioned into processes of sufficiently large grain-size.

Keywords: parallel simulation, conservative approach, deadlock

1 Introduction

Developments in distributed computing hold promise to revolutionize scientific problem solving. By using existing hardware consisting of a variety of distributed computing resources linked by high-speed networks, the cost of computation may be reduced. Parallel simulation offers potential both in speeding up existing simulation applications and more importantly in increasing the size and complexity of models simulatable within a reasonable amount of time. Previous works on parallel simulation fall into two categories: *conservative* [6] and *optimistic* [4]. Fujimoto gave a well-covered survey on these approaches [2]. A similarity of these two approaches lies in the process-oriented methodology in partitioning a system to be simulated into *physical processes* that interact at various simulated time. The simulator consists of a set of *logical processes*, one per physical process. Logical processes are mapped onto a machine for execution based on its granularity, machine architecture, *etc.* In the *conservative approach*, causality error is avoided by ensuring that parallel events are safe to be executed before execution. The *optimistic approach* adopts a different philosophy: correcting the event execution order rather than enforcing the right order to produce correct simulation results. In the event of a causality error, a rollback mechanism annuls those events simulated ahead of time.

Recently, a window-based approach [1, 8] has been proposed. This approach requires the global information on the progress of simulation in every sub-system. It compiles all progress indicators to determine the safe distance for each one to proceed. A common critic of the conservative approach is that it may not produce good performance speedup. As to the optimistic approach there is a danger that it proceeds too fast, resulting in ravage of effort when incorrect simulation results are produced. Consequently, additional workload is incurred in undoing the errors. The main weakness of the window-based approach is that it cannot produce linear speedup because the consolidation of the global information on the progress of simulation in each sub-system is serialized.

This paper focuses on a conservative approach in parallel simulation of a multi-station queueing network (MQN) on distributed-memory multicomputers, and an analysis of its performance. We discuss the parallelization of MQN simulation using a deadlock free conservative simulation scheme. Section 2 gives an overview of the simulation model for a single server queue, and the simulation considerations in modelling a system with multiple service stations. Section 3 identifies the parallelism that can be exploited in simulating MQN, and addresses modelling and deadlock resolution issues. Section 4 describes implementation issues, the mapping of a simulation application onto a limited number of processors, and analysis of the performance results. Section 5 contains our concluding remarks and some issues requiring further investigation.

2 Simulation Models of Queueing Networks

Queues are common in our daily life. For examples, customers queue up before tellers at a bank or at check-out counters in a super market. A queue is formed whenever the demand for service exceeds the capacity to provide service in time. Besides the business scenarios cited, many applications in economics and sciences such as the traffic load of a public transport system or a telephone network may also be modelled as queueing systems. The objective of studying a queueing system, from the economics point of view, is to understand its behaviors, and then improve the service it can provide at an optimal cost. Besides mathematical analysis, the behaviors of queueing systems can also be studied using simulation.

This section discusses the queueing simulation models that run on uniprocessor machines. We first introduce the simulation model of a simple queueing system with only one service station, and discuss how the simulation is performed. Next, we extend this model to multiple service stations.

2.1 Single Service Station

The simulation model for a single service station queue (SSQ) consists of a pool of customers, server(s), waiting-line(s), an arrival process and a service process, and a clock (see figure 1). Customer and server are generic terms where customer refers to any "object" which requires service, and server is any "agent" which provides the desired service.

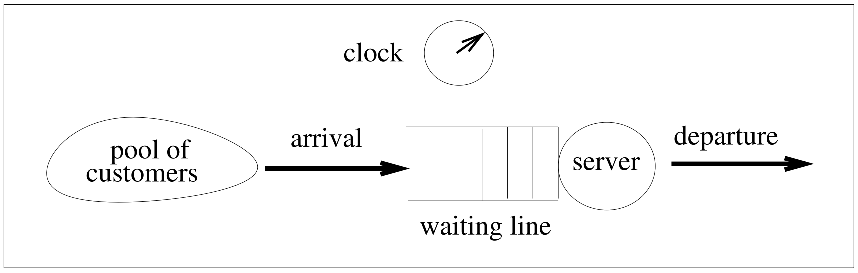


Figure 1: A queueing model for one service station

The size of the waiting line delimits the number of customers may be queueing in the system. The arrival and service processes are governed by interarrival time and service time respectively, which may be constants or variables following certain statistical distributions. The clock indicates the progress of simulation. In discrete-event simulation, the execution of each event changes the states of the system such as the queue length, waiting time, server utilization and others. The two types of event are *arrival* and *departure*. Each event is tagged with an occurrence time and the event executions must be scheduled in ascending time order to produce a correct simulation result. An illustration of such a constraint is as follows. Take two events A1 and D3, where A1 is an arrival event which corresponds to a customer coming to a service station at 1:00pm, and D3 is a departure event which corresponds to the customer leaving the service station at 3:00pm. It is obvious that the occurrence, and therefore the execution sequence of the events, of A1 must be before D3. Otherwise, it absurdly means that the future is before the past. We henceforth refer to the error caused by incorrect execution sequence as a *causality error*.

Go back to the above example for single service station. Even after A1 is executed, the execution of D3 will also cause causality error if the second arrival, A2, occurs at 2:00 pm. The reason is that in the real system the second customer will have to wait in queue on its arrival as the first customer will not leave the service station before 3:00 pm. If the events are execute in A1, D3, A2 sequence, meaning that the first customer has already left the service counter when the second customer arrives, the executions of D3 and A2 will update the system state erroneously.

There are many event scheduling schemes to avoid causality error. We discuss two of them here. For simplicity of discussion, we assume that there is only one server in the service station and the size of waiting line is infinite. In both schemes the execution of an arrival event generates the next arrival to ensure that a late departure event is not executed before an early arrival event. To ensure that every arrival will finally leave the system, each of the arrival events must have one departure event corresponding to it. The generation of departure events varies slightly in both schemes. The first scheme illustrated in figure 2 generates a departure event whenever an arrival event is executed, while the execution of a departure event only updates the system state. In the second

```
increment the number of arrivals by 1;
schedule the next arrival event;
schedule a departure event for
  this arrival;
if server is idle
   set server to busy;
else
   increment queue length by 1;
```

(a) Arrival Event

```
if queue length is 0
   set server to idle
else
   decrement queue length by 1;
```

(b) Departure Event

Figure 2: Events scheduling for one service station (scheme 1)

scheme shown in figure 3, the execution of an arrival event will generate a departure event only if the server is idle, and the execution of an departure event will have to generate another departure event if they are customer(s) waiting in queue. Obviously,

```
increment the number of arrivals by 1;
schedule the next arrival event;
if server is idle
   begin
      set server to busy;
      schedule a departure event for
        this arrival;
   end
else
   increment queue length by 1;
```

(a) Arrival Event

```
if queue length is 0
   set server to idle
else
   begin
      decrement queue length by 1;
      schedule the next departure event;
   end;
```

(b) Departure Event

Figure 3: Events scheduling for one service station (scheme 2)

the first scheme is simpler as compared to the second one. Nevertheless, if the service rate of the queue is slower than the arrival rate, the first scheme will create a long list of departure events waiting for execution. As for the second scheme, it guarantees that there are at most one arrival event and one departure event pending at all times. We adopted the second scheme in this paper. The main control of the simulation is depicted in figure 4. It is basically a discrete-event simulation algorithm where the progress of clock corresponds to the event occurrence time. The simulation repeatedly selects an event with the least occurrence time from all pending ones to execute. It terminates when the clock passes the duration of simulation.

```
set clock to 0;
set server to idle;
set queue length to 0;
schedule the first arrival event;

repeat
   select the next event with the smallest occurrence time from outstanding event(s);
   update the system state in the time interval [clock, next event time];
   advance clock to the next event occurrence time;
   execute the next event according to its event type;
until clock ≥ duration of simulation;
```

Figure 4: Main control of simulation for one service station

2.2 Multiple Service Stations

The simulation model for MQN is based on the previous model for SSQ. We assume that service stations are connected in a pipeline as shown in figure 5. We shall now

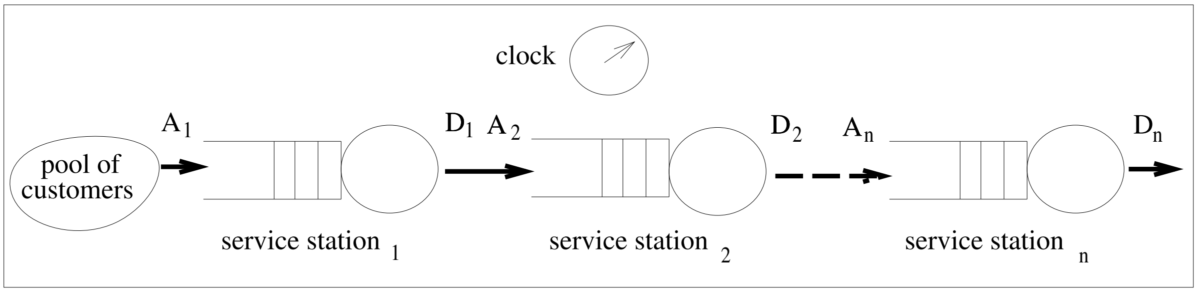


Figure 5: A queueing model for multiple service stations

formalize some notations for describing MQN simulation. Let n be the number of service stations on the pipeline. Let A_i and D_i, $1 \leq i \leq n$, be the arrival and departure events respectively for the ith service station. The system state of the queueing model comprises n sets of sub-state, where each set corresponds to the local state of a service station. The main control of the simulation is similar to that for simulating a SSQ (see figure 4), except that there may be up to $2n$ outstanding events when simulating a MQN. The simulation starts with the initialization of a clock and n sets of sub-state. It then schedules the first arrival event for the first service station. In each of the iterations, the selection of the next event for execution is from the outstanding events in all service stations. The bookkeeping of the system state includes updating the n sets of sub-state. The scheduling of events varies slightly as compared to that for a SSQ. In the MQN simulation, events A_i and D_i will only update the ith sub-state. The pseudo-code for executing each A_i remain unchanged (see figure 3a). As for each of the departure events D_k, $1 \leq k \leq n-1$, it should generate an arrival event A_{k+1} with the same occurrence time for D_k, in addition to the pseudo-code depicted in figure 3b. This corresponds to the real system where customers leaving the kth station will enter the $(k+1)$th station. The pseudo-code for D_n remain unchanged (refer to figure 3b) as it is a departure event in the last station. The progress of the system time and the termination condition for simulating MQN are similar to those for SSQ.

3 Parallel Simulation of MQN

Although the simulation model in section 2.2 may serve as a tool to analyze the behaviors of a MQN, it becomes inadequate when the number of stations is scaled up. It is clear that for c customers to enter a pipeline containing n stations, the number of events to be executed is approximately $2 \times c \times n$, meaning that the time required to run the simulation is linearly proportional to the size of the pipeline. Such a performance is certainly not desired for large scale simulation.

3.1 Parallelism in a MQN

An alternative to reduce the simulation runtime is to exploit the parallelism in MQN, and run it on multiprocessor machines. We now include additional notations for the subsequent discussion. Let A_i^j, $1 \leq j \leq c$, be the jth arrival event in the ith station. Let $t(A_i^j)$ and $t(D_i^j)$ be the occurrence time of A_i^j and the departure time for D_i^j respectively. τ denotes the duration of simulation.

Indeed parallelism is observed in a MQN. For example, the arrivals and departures from different service stations may occur simultaneously. Therefore the queueing simulation can be performed in parallel. Take five events A_1^1, D_1^1, A_2^1, D_2^1 and A_1^2, where $t(A_1^1) = 1$, $t(D_1^1) = 3$, $t(A_2^1) = 3$, $t(D_2^1) = 4$ and $t(A_1^2) = 5$. It is clear that the execution of A_1^2 must be after D_1^1 so as to avoid causality error. However, the simulation results still remain correct if A_1^2 is executed together, or even before, D_2^1 (or A_2^1) although such an execution sequence does not conform to the event occurrence times. The reason is that A_1^2 and D_2^1 (or A_2^1) update different sets of sub-state, and therefore their side-effects are mutually exclusive. In general, the side-effects of events belonging to different service stations are mutually exclusive, and it follows that these stations can be simulated in parallel.

3.2 Modelling MQN on Distributed-Memory Machines

Due to the mutual exclusion of all sub-states in a MQN, the simulation is best suited to run on distributed-memory machines. Each station can be modelled as a process. In the real system all service stations operate autonomously and so does its counterparts in simulation. Communication among the processes can be performed by passing messages. Take a n-station pipelined queue for example. We can model the system using $n+1$ processes, denoted by $G, P_1, P_2, ..., P_n$. Figure 6 shows the interprocess connection and the direction of message flow on the pipeline. All the $n+1$ processes have their own clock. The process G serves as a customer generator and sends arrival messages to P_1. Each of the P_i performs the queueing simulation only for one station. Except for the last process P_n, each P_i also sends arrival messages to P_{i+1} when it executes departure events. Since all the $n+1$ processes may be executed in parallel, the parallel simulation runtime is approximately $2c$ if the number of processors available, denoted by p, is $n+1$. It follows that the efficiency of the parallel simulation, excluding communication

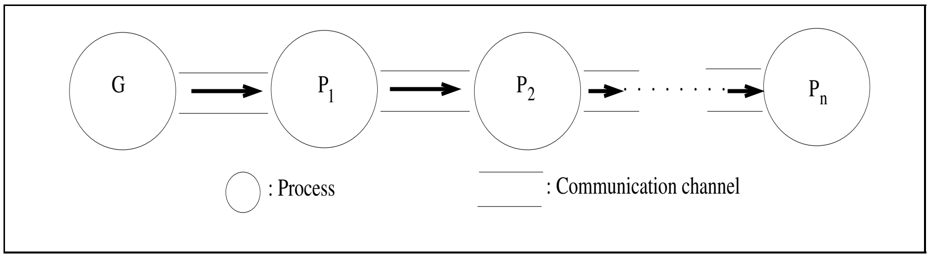


Figure 6: Interprocess connection and message flow

overheads, is $\frac{2nc \times 1}{2c \times (n+1)} \times 100\%$, which is asymptotic to 100% for large n.

To avoid causality error in each process P_i, a conservative approach is used in the parallel simulation. Each process repeatedly selects an event with the smallest occurrence time and executes it according to its event type (refer to figures 3a, 3b and 4). In the parallel simulation, when executing an arrival event a process need not schedule the next arrival event. Instead, it waits for an arrival message, containing the arrival time of the next customer, from its in-coming channel(s). Each of the processes terminates when its own clock passes τ.

3.3 Resolving Deadlock in MQN Simulation

It is noteworthy that although the conservative technique produces the correct simulation results, deadlock may occur in some processes, especially when the clocks are closed to the completion time. Consider the following scenario. Suppose $\tau = 8$ hours, $t(A_1^1) = 7$, $t(D_1^1) = 7.5$, $t(A_1^2) = 8$, and $t(D_1^2) = 8.5$. The second event executed by P_1 will send an arrival message, corresponding to A_2^1, to P_2. When P_2 executes this arrival event, it will have to wait for the next arrival sent by P_1 so as to prevent causality error. Unfortunately, P_1 will not send the next arrival message because $t(D_1^2)$ is beyond the simulation duration. In fact P_1 only executes the first three events and terminates, causing deadlock in P_2. Consequently, all the succeeding processes are also deadlocked. There are many proposals to resolve this type of deadlock problem. One of them is that whenever a process, say at time θ, has nothing to send it will send *null messages* on all its out-going channels, informing the receiving processes not to wait until time $\theta + \epsilon$, where ϵ is the minimum transit time required by the customers in the sending process. However, such a mechanism fails if a cycle exists in the system topology [5]. Other proposals even allow deadlock to occur, but have some mechanisms to break the deadlock [7].

The way we resolve the deadlock problem is simpler. Each of the processes G, P_1, P_2, ..., P_{n-1}, will have to intentionally send out a *pseudo-arrival message*, containing an arrival time of $\tau + \delta$, $\delta > 0$, to its successor before it terminates. When a P_i picks an event of occurrence time greater than τ, it will only update the ith sub-state in the time interval [clock, τ]. Next, it sends a pseudo-arrival to P_{i+1}, and terminates itself subsequently. Therefore such an event will only serve as a delimiter for simulation in the receiving process, instead of being executed. Note that such actions may also happen on a departure event of occurrence time greater than τ. For that circumstance the process will terminate before the arrival of pseudo message. Nonetheless, the operating system will deal with such pseudo message(s) during garbage collection.

We shall now illustrate how such a scheme resolves the deadlock problem in P_2. Let T_i be the clock in the P_i. In our resolution, P_1 will send the pseudo-arrival, corresponding to A_2^2, to P_2 before it terminates, in addition to the first arrival message, corresponding to A_2^1, sent at $T_1 = 7.5$. In P_2, the arrival event A_2^1 is executed when $T_2 = 7.5$, and the wait for the second arrival event A_2^2 is no longer infinite. A departure event D_2^1 will be generated when executing A_2^1. If $t(D_2^1) < 8$, the last event selected by P_2 will be A_2^2. Otherwise the last event is either D_2^1 or A_2^2, depending on which event is of a smaller occurrence time. In either case, the last event will only serve as a delimiter of simulation in P_2 if the event occurrence time is greater than 8. Similarly, P_2 will also send a pseudo-arrival message to P_3, thereby avoiding the occurrence of deadlock in other processes.

Unlike the null message mechanism, which may overload the communication channels, our resolution will only transmit one pseudo-arrival before the end of the execution in process G, and the first $n-1$ P_is, amounting to only n additional messages in a parallel simulation run.

4 Implementation Details

We have implemented the parallel simulation algorithm in the C language. It runs on a network of workstations which mimics a distributed-memory parallel computer. Before we present the performance results, we briefly describe our implementation platform, and illustrate how we map the simulation program onto the processors in the network.

4.1 Implementation Platform

PVM (Parallel Virtual Machine) is a software package that links a heterogeneous network of UNIX-based parallel and serial computers to work as a single concurrent computer resource [3]. PVM provides facilities for spawning, communication, and synchronization of processes over a network of heterogeneous machines. It allows its user to create processes on its own and other computers, and to send messages. The PVM system consists of two parts. The first part is a daemon process, called *pvmd*3, which must be spawned in all computers making up the virtual machine. It's main task is to handle the message communication among processes located in the same and on different computers. The second part of the PVM system is a library of interface routines for passing messages, spawning processes, coordinating tasks and modifying the virtual machine. Before executing the simulation program, we have to configure the virtual machine. It is done by running a software utility, called *pvm*, to spawn the daemon on each workstation to be included in the network.

4.2 Mapping Simulation Program to Virtual Machine

For a n-station queue, the parallelism in the simulation can be fully exploited provided $p \geq n+1$. However, in actual cases the number of processors available is always confined to a small number. Therefore some workstations (processors) will have to execute more than one processes during the simulation run.

Two aspects, load balancing and communication overhead, are considered when we map the simulation program to the virtual machine. In the first aspect, we note that, except P_n, the workload of each P_i are identical. The workload of P_n is slightly smaller than those of the other P_is as it does not have to send out arrival message when executing departure events. The workload of G is smaller as compared to those of P_is as its task is just to generate arrivals and send them to P_1. Therefore, the inclusion of G on any workstation will not severely disturb the load balance. In reducing communication overhead, a simple strategy is to group those processes which communicate close to each other in a processor. Since the service stations are connected as a pipeline, the placement of processes on processors becomes straightforward. Figure 7 shows an example of the mapping where $p < n+1$. The PVM programs for the customer generator process and

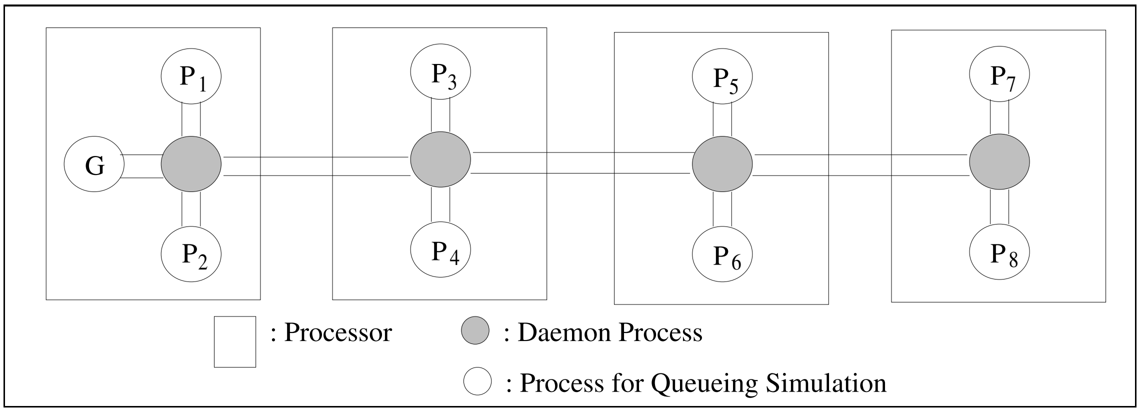


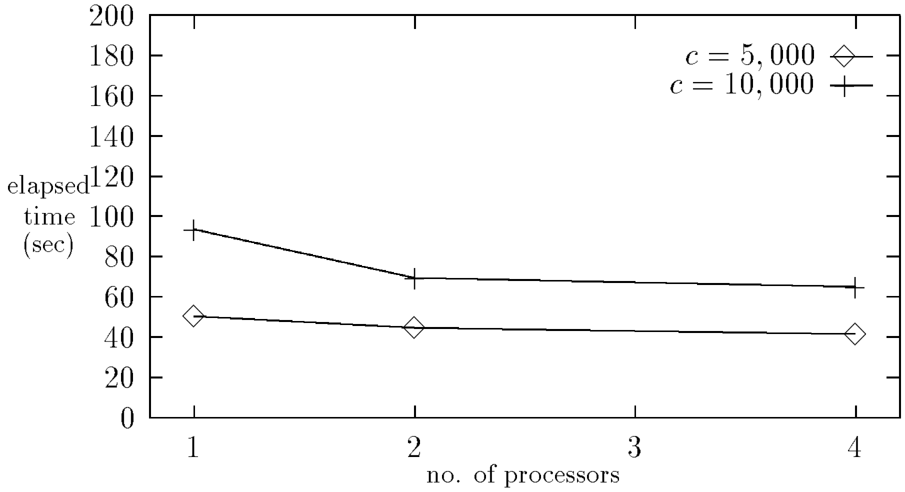
Figure 7: Processes to processors mapping for a pipelined queue ($n = 8, p = 4$)

a server process can be found in the appendix.

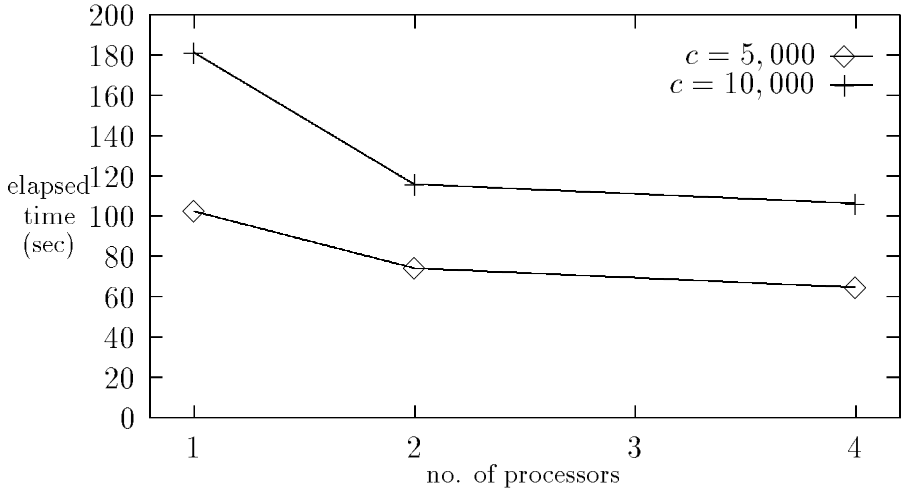
4.3 Performance Analysis

The performance of the conservative simulation program was evaluated using four work-stations running PVM. Elapsed time and CPU time are captured. Due the the multi-user and multi-tasking operating environment, and the fact that workstations constituting the virtual machine are housed in different locations, the communication overheads are high. Therefore, in the following analysis the performance of the parallel simulation will be gauged by its trends, instead of the individual timings. Exponential distribution is used to generate the interarrival time and service time. As convergent results are obtained for different set of simulation parameters, we will only present the timings obtained from c = 5,000 and 10,000, and n = 4 and 8, and p = 1, 2 and 4.

Figure 8 shows that as the number of workstations increases, the elapsed time of the simulation decreases. Such a phenomenon shows that the execution time of the queue-



(a) 4 service stations



(b) 8 service stations

Figure 8: Elapsed Time versus No. of Processors

ing simulation can be reduced by using more processors. However, it is important to highlight that adding beyond $n + 1$ processors will not further reduce the elapsed time as there is no more process to be allocated to the additional processors.

Figure 9 shows that the CPU time required by the parallel simulation decreases as more processors are added. In particular, the decrease in the CPU time is proportional

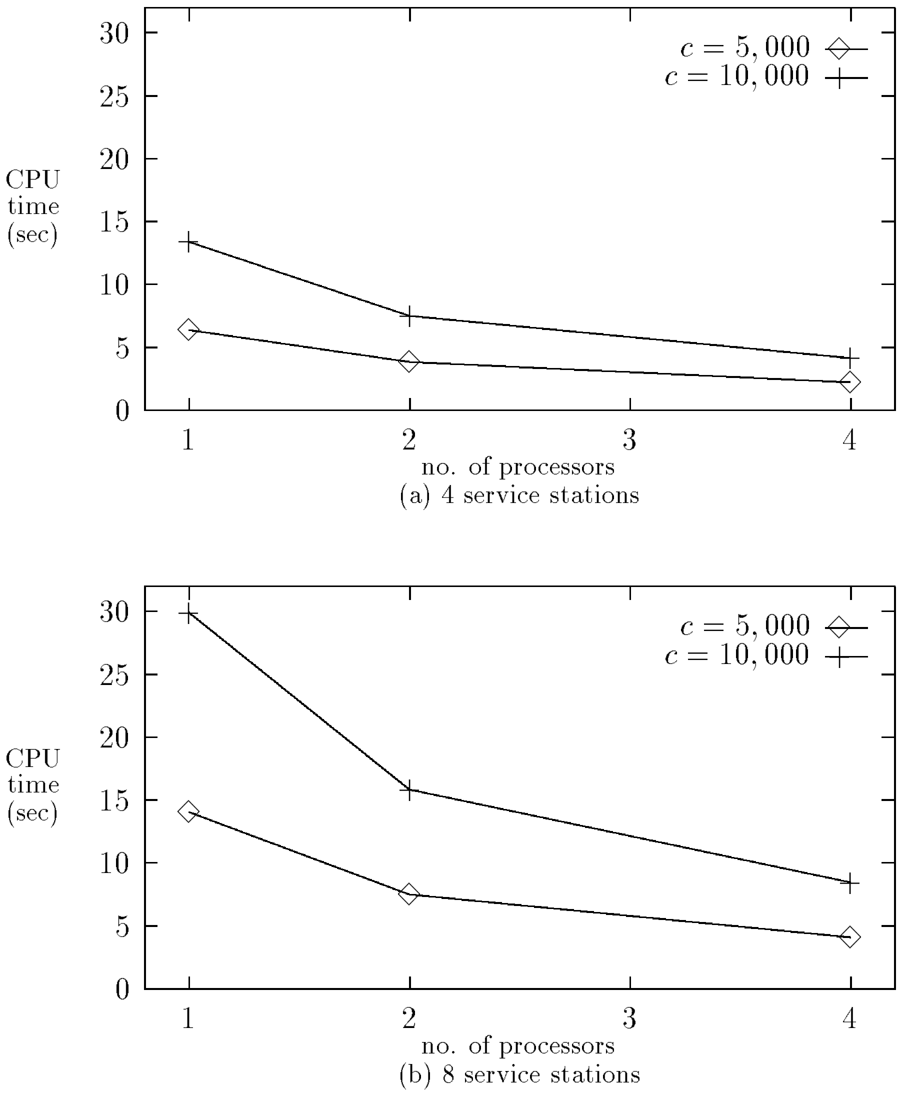


Figure 9: CPU Time versus No. of Processors

for high workload when the number of workstations increases from 1 to 2 (figure 9b, $c = 10,000,\ n = 8$).

From figures 8b and 9b, we assert that if the workload in each process outweighs its communication overhead, the speedup of the simulation will be close-to-linear. In a separate experiment conducted to verify the assertion, the workload in each process was gradually increased to observe the changes in the elapsed time and the simulation speedup. Figures 10 and 11 show the elapsed times and the corresponding speedups respectively for different volumes of workload. The parameters used in the investigation were $c = 100$ and $n = 4$. Let ξ be the addition workload per event. In the investigation ξ is simulated by a nested *for loop* with runtime ranging from 6ω to 40ω, where $\omega \approx 8 \times 10^{-3}$ sec. Figure 11 shows that as the workload to execute an event is increased so is the

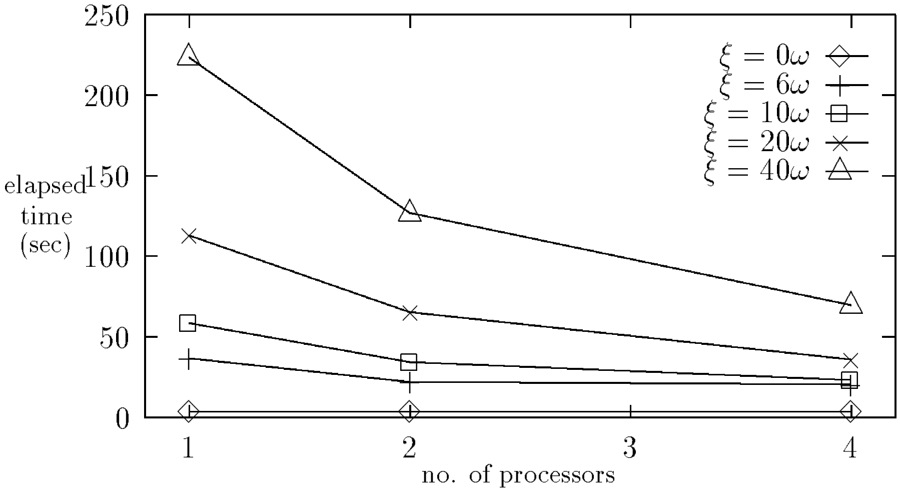


Figure 10: Elapsed Time versus No. of Processors

speedup of the simulation. Such an observation shows that a close-to-linear speedup is attainable even when blocked-receive is adopted in the conservative simulation, provided the workload in each process is high. The CPU utilization of the conservative simulation for more processors can be observed in figure 12. Let η be defined as the ratio of communication overhead and CPU time. This measure shows that the smaller the value of η, the better is the CPU utilization for the parallel simulation. We observe from the figure that η decreases for more processors and more workload per event, meaning that the utilization of the CPU is improved, or the communication overhead becomes less significant, when the granularity of parallelism is coarse and the number of processors used is increased.

5 Conclusions and Further Works

We have shown how simulation of queueing networks can be performed in parallel using a conservative approach that exploits the computing resources of a cluster of workstations. A new and more efficient approach for avoiding deadlock using *pseudo-arrival messages* is proposed. Despite the fact that blocked-receive is adopted in the conservative approach, we have shown that a close-to-linear speedup is attainable if the system to be simulated can be partitioned into a sufficient number of coarse-grain logical processes.

We are currently extending the proposed conservative scheme to handle more complex queueing topologies, such as multistage interconnection networks so as to study the efficiency of conservative simulation with respect to network topology. In this respect, partitioning/mapping/scheduling strategies for efficient execution of logical processes are needed. As we reach the limit in exposing conservative simulation parallelism, further improvement in execution speed will depend on exploiting speculative simulation parallelism. A hybrid scheme that exposes both types of parallelism is being investigated.

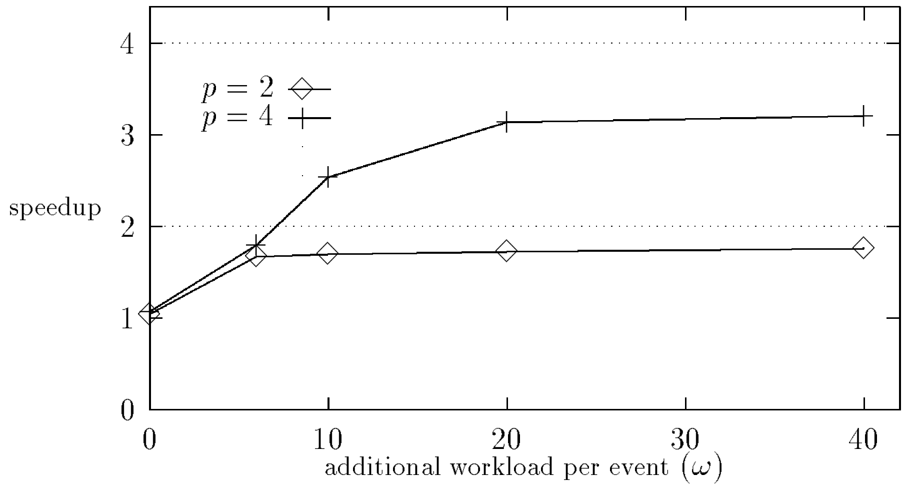


Figure 11: Speedup versus Additional Workload per Event

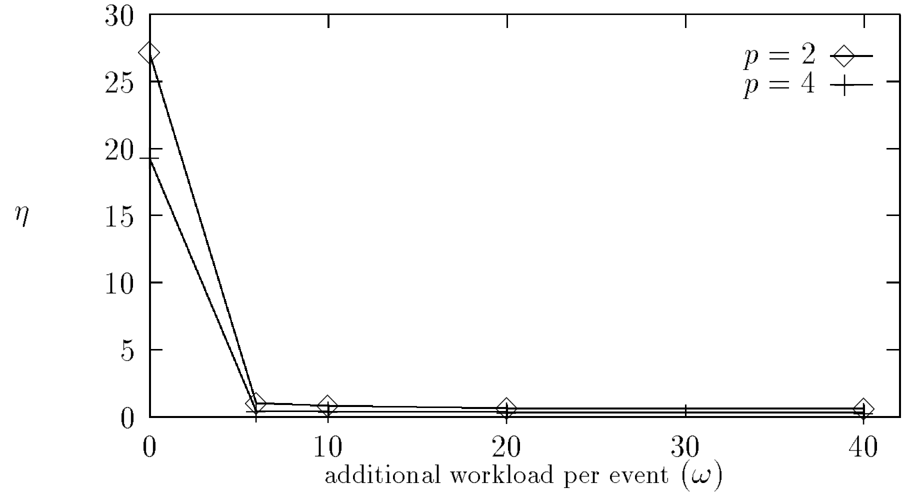


Figure 12: Overhead Ratio versus Additional Workload per Event

References

[1] R. Ayani and H. Rajaei, *"Event Scheduling in Window Based Parallel Simulation Schemes"*, Proc. 4th Symposium on Parallel and Distributed Processing, pp. 56-60, 1992.

[2] R.M. Fujimoto, *"Parallel Discrete Event Simulation"*, Communication of ACM, Vol. 32, No. 10, pp. 31-53, October 1990.

[3] A. Geist, et al., *"PVM 3 User's Guide and Reference Manual"*, Oak Ridge National Laboratory, Tennessee, ORNL/TM-12187, May 1993.

[4] D.R. Jefferson, *"Virtual Time"*, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3., pp. 404-425, July 1985.

[5] R. Konas and P. Yew, *"Parallel Discrete Event Simulation on Shared-Memory Multiprocessors"*, Proceedings of the 24th Annual Simulation Symposium, pp. 134-148, Louisiana, April 1991.

[6] J. Misra and K.M. Chandy, *"Asynchronous Distributed Simulation via a Sequence of Parallel Computations"*, Communication of the ACM, Vol. 24, No. 4, pp. 198-206, April 1981.

[7] J. Misra, *"Distributed Discrete-Event Simulation"*, Computing Surveys, Vol. 18, No. 1, pp. 39-65, March 1986.

[8] Y.M. Teo and S.H. Tan, *"Parallel Discrete-Event Simulation based on Multiple Time Windows"*, Proceedings of the European Simulation Symposium, Delft, The Netherlands, The Society for Computer Simulation International, pp. 359-364, October 1993.